A New Approach to Vector Code Generation for Applicative Languages

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331
budd@cs.orst.edu

September 20, 1994

Abstract

In an earlier paper we developed an intermediate representation for languages based on composition, and showed how the representation could facilitate generating code for functional languages, such as FP. In this paper we follow the same philosophical approach, using instead the applicative language APL. Further, we show how this intermediate representation simplifies the task of generating code for highly parallel machines, such as the Connection Machine.

keywords: *code generation, vector machines, applicative and functional languages*

1 Introduction

In an earlier paper [Bud88b] we introduced an intermediate representation specifically designed for use with languages based on composition. Examples of such languages include functional languages, such as FP [Bac78], and applicative languages, such as APL [Ive80]. We then argued that it was not only easy to convert from such languages into this representation, but also easy to generate actual machine code from the representation. The earlier paper presented examples showing the utility of the approach using the language FP and generating code for a conventional processor. In this paper we will follow the same philosophical approach, but vary both the front end language and the type of processor for which we are producing code.

Section 2 will review the internal representation we use in our code generation scheme, and describes how the representation for APL differs slightly from that used in the FP example. Section 3 analyzes an example APL expression, showing how it is converted into this internal form. Section 4 then shows how this internal form can be used to produce code suitable for execution on vector machines.

2 Representation

Our internal representation for APL arrays is slightly more general than the representation described in [Bud88b] for FP style lists. This is due to the necessity of recording not only the number of elements in the list (the list length), but the rank and shape of the data as well. As in the FP representation, our representation will consist of a pair, the second element of which is a function (written in λ notation), for producing an arbitrary value of the result. Values are indexed starting at zero. The first element is also a pair, representing the rank (a scalar), and the shape (a vector). In all cases these can be arbitrary expressions.

For example, a scalar constant, such as 200, is represented as follows:

$$((0,\ 1),\ \lambda\ \mathrm{p}\ .\ 200\ )$$

Here the rank is zero (that is, a scalar). Since the shape is a vector of length zero, its value is really of no importance in this case; we use the constant one merely as a convention. The value is given by a function which will in all cases return 200.

As in the earlier paper we will use a dot-field notation for extracting the various parts. The first field, the rank-shape pair, will be denoted by the field name *size*. The individual parts will be denoted by the names *rank* and *shape*, respectively. The value field will be denoted by the field name *item*.

Functions are represented by λ expressions of the appropriate arity which take and return a representation as above. We will use the APL iota function as an example. This function takes a single scalar argument and returns a vector, the length of which is determined by the value of the argument. The values of this vector range from 1 upwards. For example, $\iota 5$ yields the vector 1 2 3 4 5. The iota function can be represented as follows:

$$\iota \equiv \lambda\ \mathrm{a}\ .\ (\ (1,\ \mathrm{a.item}\ 1),\ \lambda\ \mathrm{p.p{+}1})$$

This function takes a single argument. It returns a scalar, as indicated by the constant 1 in the rank field. The size of the scalar is given by the value in the first position of the argument. when the p^{th} value is requested, the value $p+1$ (reflecting that values are indexed starting with zero) is returned.

As in the earlier paper we will simplify the initial discussion by ignoring the issue of argument conformability checking; although there are no intrinsic problems involved in that issue. We will also introduce some new notation as we go along.

3 Presentation of an Example

As was discussed in the introduction, we will in this paper only present a description of how we would go about generating code for a single specific APL expression. Thus we will only describe the code generation technique necessary for functions used in that one expression. A more complete description of the internal representation of APL functions can be found in [Bud88c].

The APL expression to be analyzed is the classic idiom for producing a bit-vector of size N, where the position of the 1 values correspond to the primes larger than two [PeR79].

$$2 = +\not.0 = (\iota N) \circ . \mid \iota N$$

Thus the elements to be described are:

- The scalar variable N
- The constants 0 and 2. (The characterization of constants was given in section 2).
- The iota function (which we have also already seen in section 2).
- The dyadic scalar functions divides, plus and equals.
- the outer product operator.
- The reduction operator.

We will provide only a brief description of the meanings of the various APL functions. For a more complete description of the language see, for example [PoP75].

3.1 Characterization of Variables

The code to be generated for variables differs depending upon whether it is known if a variable is a scalar or not. Although APL does not require declarations of any sort, several implementations permit optional declarations of variable rank in order to assist the compiler [Bud88a],[Syb80]. Even where such declarations are not present, simple dataflow analysis can often be used to determine this information [Bud88a],[Str77].

If a variable, say N, is known (either by declaration or inference) to be a scalar, then the internal representation is as follows:

$$((0,\ 1),\ \lambda\ \mathrm{p}\ .\ (\mathrm{N.value}\ 1))$$

If the rank is known to definitely not be a scalar, then the code is as follows:

$$(\mathrm{N.size},\ \lambda\ \mathrm{p}.\ (\mathrm{N.value}\ \mathrm{p}))$$

It will simplify the support of scalar extension (to be discussed in the next section), if when nothing is known about the rank of N, we generate code which will test to see if the value is a scalar on each access, and if so return the scalar value.

$$(\mathrm{N.size},\ \lambda\ \mathrm{p}\ .\ \text{if } 0 = \mathrm{N.rank} \text{ then } (\mathrm{N.value}\ 1) \text{ else } (\mathrm{N.value}\ \mathrm{p}))$$

3.2 Characterization of Dyadic Scalar Functions

The language APL generalizes the standard dyadic functions, such as addition (+) and subtraction (−) to apply pairwise to elements of arrays. For example:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} + \begin{array}{ccc} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{array} = \begin{array}{ccc} 2 & 6 & 10 \\ 6 & 10 & 14 \\ 10 & 14 & 18 \end{array}$$

In addition, a feature known as scalar extension allows scalar quantities to be used in operations with arrays. In situations where this occurs the scalar is treated as if it were an array of the appropriate size, containing the scalar value repeated as often as necessary. For example

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} + 2 = \begin{array}{ccc} 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{array}$$

The only scalar extension which occurs in our example concerns scalar constants. The representation for such constants was given in section 2, from which one can note that the index for the particular element being requested is ignored. Thus in this case scalar extension costs (in terms of code) almost nothing.

3.3 Characterization of Outer Product

Outer product in APL is a functional; that is a function which takes another function as argument and yields a third function, which is finally applied to data values. In the companion paper on FP we presented several examples of such functions, such as the functional *apply* which took a function and returned a function which when presented with a list applied the original function to each element of a list. In contrast to FP, where the argument function given to a functional could be arbitrary, the functions that can be used in an outer product are restricted to just the primitive dyadic scalar functions (such as addition or subtraction). Thus in order to simplify the presentation we will merely discuss the characterization of an outer product of a specific dyadic function, and not the general case.

Outer product takes two array arguments, and returns the array produced by applying the dyadic scalar argument given as part of the outer product to each possible pairing of values. In our example expression the arguments are both vectors produced by the subexpression ι N. The vertical bar is the APL remainder function, which returns the result when the second argument is divided by the first. Thus if N is 5, for example, the result of $(\iota N) \circ . | \iota N$ is the following two dimensional array:

$$\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 2 & 0 & 1 & 2 \\ 1 & 2 & 3 & 0 & 1 \\ 1 & 2 & 3 & 4 & 0 \end{array}$$

The rank of the result of an outer product is the sum of the ranks of the arguments, and the shape is the catenation of the shapes of the arguments. The computation of a specific element p of the result can be given in terms of the size (number of elements) in the left argument (call it a). It is the quantity produced by the scalar function on the element found in the left argument at position p div sizeOf(a) and the right argument at position p mod sizeOf(a). Thus we can characterize the outer product as follows:

λ a,b ((a.rank + b.rank), concat(a.shape, b.shape)),
    λ p. (a.item (p div sizeOf(a))) *op* (b.item (p mod sizeOf(a))))

where sizeOf() computes the size of the left argument (a quantity which may be known at compile time) and *op* is the primitive dyadic scalar function.

There is one very important difference in approach that should be noted between the technique described here and that used by the scalar APL compiler [Bud88a]. In the scalar compiler it was noted that certain expressions, such as p div sizeOf(a), would repeatedly produce the same answer for a succession of values. Thus explicit control flow constructs were introduced to avoid this recomputation whenever possible. In the present case, the underlying assumption is that these values may possibly be generated in parallel. The parallel evaluation of a large number of expressions which will all result in the same value is of no consequence, and is in fact preferable to a single processor producing the value and distributing it to all the others. We will make this point again in section 4 when we discuss the code generated for this expression.

3.4 Characterization of Reduction

The reduction operator in APL is used to reduce a sequence of values to a single result; for example computing row or column sums of an array. As with the insertion functional of FP, this operator involves an implicit loop. The loop is not explicitly given in our internal representation; rather, in order to permit the code generator as much flexibility as possible, we encode enough information to permit the reconstruction of the loop. This permits us to more easily make manipulations of the reduction form, and it also allows the code generator freedom to perform actions such as computing multiple values of the loop in parallel.

In order to better understand exactly what information needs to be recorded in order to generate code which will perform the reduction, let us consider a simple example. Consider computing the column sums of a three by five array. Regardless of dimensionality, values in memory are stored in sequential order (called *ravel* order in APL). Thus the index positions of this array are as follows:

$$\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 \end{array}$$

The column sums will be a vector of five elements; the first of which is formed by summing the values in the positions 0, 5 and 10, the second by summing values from positions 1, 6 and 11, and so on.

In order to know exactly which elements need to be summed for any particular value, three quantities are important. The first, the *extent*, is the length of the column which will be summed. In our case the extent is 3. The second quantity, the *delta*, is the difference in index values between successive elements of the column. If our example this difference is 5 (for example, position 2 is added to position 2+5=7). The final value is the *resultSize*, which is the length of the resulting value.

If we were computing row sums instead of column sums the values for *extent*, *delta* and *resultSize* would have been 5, 1 and 3, respectively.

We record the information necessary to reproduce this computation by introducing a new symbol, ρ, which is followed by the name of the function with which the reduction is to be performed, the three fields just described, and a λ expression representing the argument to the reduction. For

example, if the three by five array described above were contained in the variable x, the code for the reduction which computed the column sums would be as follows:

$$\rho +, 3, 5, 5, \lambda \text{ p . (x.item p)}$$

3.5 Continuation of Example

Having presented the characterization of each of the constituent parts, we now return to the question of characterizing our example expression:

$$2 = +\not.0 = (\imath N) \circ . | \imath N$$

We will form each of the compositions from the inside out. Assume first that we know from elsewhere (declarations or dataflow analysis), that the variable N is a scalar. Thus N can be characterized as:

$$\text{(N.size, } \lambda \text{ p . (N.value 1))}$$

From the definition of $\imath$ (section 2), we can compute $\imath$ N as follows:

$$(\lambda \text{ a . ((1, a.item 1), } \lambda \text{ p . p+1))) (N.size, } \lambda \text{ q . (N.value 1))}$$

$$\equiv \text{((1, } (\lambda \text{ q . (N.value 1)) 1), } \lambda \text{ p . p+1)}$$

$$\equiv \text{((1, (N.value 1)), } \lambda \text{ p . p+1)}$$

Thus the outer product $(\imath N) \circ . | \imath N$ is characterized as follows:

```
((1+1, concat((1, (N.value 1)), (1, (N.value 1)))),
      λ p . ((p mod (N.value 1))+1) mod ((p div (N.value 1))+1))
```

Leaving out some of the intermediate steps, the constant 0 and the equality test merely become part of the innermost λ expression:

```
((2, concat((1, (N.value 1)), (1, (N.value 1)))),
    λ p . 0 = (((p mod (N.value 1))+1) mod ((p div (N.value 1))+1)))
```

The reduction converts this two dimensional array back into a vector, introducing a ρ expression in the process. Again leaving out some of the intermediate steps, this becomes:

```
((1, (N.value 1)),
    λ p . ρ +, (N.value 1), (N.value 1), (N.value 1),
              λ q . 0 = (((q mod (N.value 1))+1) mod ((q div (N.value 1))+1)))
```

Finally the constant 2 and the equality test, after some manipulations, merely slip in under the outermost λ, yielding our final characterization of the expression.

```
((1, (N.value 1)),
    λ p . 2 =
        ρ +, (N.value 1), (N.value 1), (N.value 1),
             λ q . 0 = (((q mod (N.value 1))+1) mod ((q div (N.value 1))+1)))
```

Notice that as we saw with the representation of FP expressions [Bud88b], the size (in terms of the number of characters), of the compositions grows very slowly as more and more terms are added. The size of the resulting expression is far less than the sum of the sizes of the individual elements. However the most important characteristic of this representation is the ease with which it permits the generation of code for a vector processor, which is the topic of the next section.

4 Code Generation for a Vector Machine

In [Bud88b] we showed how to generate code for a conventional scalar processor from a representation very similar to the one used here. In this section we will consider the slightly different question of generating code for machines which possess the ability to act on entire vectors of values in one operation. Examples of such machines are the Crays, the CDC-205, or the Connection Machine [Hil85].

In the algorithms presented in [Bud88b] for generating code for a scalar processor, both λ and ρ expressions became loops, which computed their values one by one. The code generated for vector machines will be quite different, as a goal will be to compute as many quantities in parallel as possible. Our notation will be similar to that of Hillis and Steele [HiS86]. We will assume the existence of an instruction

for all k **to** n **in parallel do** s **od**

which will execute the statement s in parallel n times, each instance of s having access to a unique value k. The total time for this computation will be assumed to be the same as the time necessary for a single execution of s. Our notation differs from Hillis and Steele not only in giving explicitly the upper bound n, but also we will assume the values of k run from 0 to $n-1$, instead of from 1 to n.

Given this framework, code generation for λ expressions becomes straightforward. A λ expression can compute all of its elements in parallel in a single step. For example, the innermost λ computation in the example presented in section 3 is as follows:

$$\lambda \text{ q } . \text{ } 0 = ((\text{q mod (N.value 1)})+1) \text{ mod } ((\text{q div (N.value 1)})+1)$$

If we assume the common subexpression (N.value 1) has been placed into the scalar variable n, then from this we will generate the following code, which places the result into the temporary variable x.

for all q **to** $n * n$ **in parallel do**
    $x[q]$ **:=** $(0 = (((q \textbf{ mod } n) + 1) \textbf{ mod } ((q \textbf{ div } n) + 1)))$
od

In no more time than it takes to do two additions, three divisions and an equality test, all $n * n$ values are produced.

Code generation for reduction expressions (ρ expressions) is slightly more complicated. Recall that each element of the reduction is formed by processing a column from an underlying expression. In order to compute the index of the elements of this column the ρ expression maintains, in addition to the function with which the reduction will be computed, three additional values. These values

are the *delta*, the distance (in the underlying expression) between adjacent elements of the same column, the *extent* (length) of a column, and the *resultSize*, which is the number of elements that will eventually be generated by the reduction. Using these three values we first generate a loop which assigns a number, the column position number, to each element in the underlying expression.

```
for all q to resultSize * extent in parallel do
        cp[q] := (q div delta) mod extent
od
```

There are two optimizations that should be noted. When reduction is along the first dimension, as in this example, *delta* is equal to *extent* and thus the **mod** can be eliminated. When reduction is along the last dimension, *delta* is equal to one and the **div** statement can be eliminated.

For each element in the argument to reduction, the value *cp* indicates the offset of the element in its associated column. To actually perform the reduction, the algorithm is a variation of Hillis' and Steels' log N array summing algorithm[HiS86][1]:

```
for j := 1 to ceil(log2 extent) do
    for all k to resultSize * extent in parallel do
            if cp[k] mod 2^j = 0 then
              if cp[k] + 2^(j-1) < extent then
                 x[k] := x[k] op x[k + (2^(j-1) * delta)]
              fi
            fi
    od
od
```

Note that the execution time of this code is proportional to the ceiling of the log base two of the size of the column being summed. When reduction is not along the first dimension (as it is in the present case), the results of this computation are not, however, in their proper place. Instead, they are scattered throughout the variable x. In those cases it is necessary to generate one further statement in order to move the values to their correct location at the beginning of the array.

```
for all k in resultsize * extent in parallel do
        if cp[k] = 0 then
          x[(k div (extent * delta)) * delta + k mod delta] := x[k]
        fi
od
```

On the Connection Machine [HiS86], there is a special instruction which makes this loop somewhat simplier. The pseudo-variable **enumerate** returns one less than the number of processors which are active and have index values less than the current processor. Therefore this loop can be written as:

[1] We note that Hillis' and Steels' algorithm is only correct if the function being used for the reduction is associative. The language APL actually allows reductions with nonassociative operators, such as division. For these cases slightly more complex, and less efficient, code must be generated. On the other hand this code does work even for functions which do not have an identity, such as maximum.

```
for all k in resultsize * extent in parallel do
        if cp[k] = 0 then
          x[ enumerate ] := x[k]
        fi
od
```

In situations where we have λ and ρ expressions nested within each other, as in our example, the code generation strategy will be to compute the inner expressions first, placing the results into temporary variables. Thus the completed code which is generated for the example of section 3 is as follows:

```
for all q to n * n in parallel do
        x[q] := (0 = (((q mod n) + 1) mod ((q div n) + 1)))
od
for all  q to n * n in parallel do
        cp[q] := q div n
od
for  j := 1 to ceil(log2 n) do
        for all k to n * n in parallel do
                if cp[k] mod 2^j = 0 then
                  if cp[k] + 2^(j-1) < n then
                    x[k] := x[k] + x[k + (2^(j-1) * n)]
                  fi
                fi
        od
od
for all k to n in parallel do
        x[k] := (2 = x[k])
od
```

The vector result in the variable x will contain a one value in each position corresponding to a prime. Despite the length of this code, it is quite efficient. The only statement which does not have a constant execution time is the loop which starts in line 7, which has running time proportional to $\log_2 n$. Thus whereas a naïve scalar implementation of this APL expression would take $O(N^2)$ operations, we have generated code to produce the result in $O(\log_2 N)$ vector instructions. If N is 100, ceil($\log_2 N$) is 7 and N^2 is 10,000, which is a difference of three orders of magnitude.

We have done this, however, by trading time for space. It is common for APL expressions to produce large values as intermediate expressions, which are then reduced or compressed to produce a small final result. Because of this, intelligent scalar APL compilers attempt to save space as much as possible during intermediate calculations [Bud88a]. Here our approach has been just the opposite; we freely allow ourselves temporary arrays of size n^2, such as the variable *cp*, if doing so will allow us to compute results in fewer vector steps. Even on a 65,536 processor Connection Machine, where each processor maintains a single value of each array, one can reach a point where this becomes impractical. It remains to be seen whether algorithms can be developed which are both space and vector-time efficient.

5 Conclusions

It is certainly dangerous to reason from examples, and thus we avoid any claim for the universality of the approach presented here. Nevertheless, we have shown that at least for the one specific expression analyzed in this paper that our intermediate representation not only facilitates the composition of complex expressions, but also simplifies the task generating code for vector machines. Given the complexities of making effective use of vector instructions in processors for languages which are basically scalar [PaW86], this approach would seem to deserve further investigation. Further work to do includes developing the internal representation characterization of the remaining APL functions, and attempting to generate code for actual vector machines.

References

[Bac78] Backus, John, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, Vol 21(8): 613-641, (August 1978).

[Bud88a] Budd, Timothy A., *An APL Compiler*, Springer-Verlag, 1988.

[Bud88b] Budd, Timothy A., "Composition and Compilation in Functional Programming Languages", Technical Report 88-60-14, Computer Science Department, Oregon State University, June 1988.

[Bud88c] Budd, Timothy A., "Characterization of APL Functions", In preparation.

[Hil85] Hillis, W.D., *The Connection Machine*, MIT Press, Cambridge, Mass. 1985.

[HiS86] Hillis, W. D. and Steele, G. L., "Data Parallel Algorithms", *Communications of the ACM*, Vol 29(12):1170-1183 (December 1986).

[Ive80] Iverson, Kenneth E., "Notation as a Tool of Thought", *Communications of the ACM*, Vol 23(8):444-465 (August 1980).

[PaW86] Padua, David A., and Wolfe, Michael J., "Advanced Compiler Optimizations for Supercomputers", *Communications of the ACM*, Vol 29(12):1184-1201 (December 1986).

[PeR79] Perlis, Alan J. and Rugaber, Spencer, "Programming with Idioms in APL", APL Quote Quad, Vol 9(4):232-235 (June 1979).

[PoP75] Polivka, R. and Pakin, S., *APL: The Language and Its Usage*, Prentice Hall, 1975.

[Str77] Strawn, G. O., "Does APL Really Need Run-Time Parsing?" *Software – Practice & Experience*, Vol 7: 193-200 (1977).

[Syb80] Sybalsky, J. D., "An APL Compiler for the Production Environment" *APL80*. North-Holland Publishing Company, 1980.